جامعة الحسن الأول
UNIVERSITÉ HASSAN 1ER

كلية العلوم و التقنيات سطات
FACULTÉ DES SCIENCES ET TECHNIQUES SETTAT

Chapitre-2 PL-SQL Oracle

Table des matières

I- Introduction :

Le langage SQL n'est pas un langage de programmation Il ne permet pas, par exemple, de définir des fonctions ou des variables, d'effectuer des itérations ou des instructions conditionnelles. SQL est un langage orienté principalement vers les opérations de recherche de données dans une base volumineuse d'une façon simple et efficace.
Il est donc clair que SQL ne suffit pas pour le développement d'applications. Dans l'utilisation courante, SQL est intégrée dans un langage de programmation Externe généraliste comme C, C++, Java, PHP, etc…. Mais cette méthode demande beaucoup d'échange client/serveur.
Des SGBD proposent une alternative à l'écriture de programmes avec une langage de programmation généraliste. Oracle propose la création des procédures écrites avec le langage PLSQL. Ces procédures stockées s'exécutent au sein du SGBD, ce qui évite les échanges réseaux qui sont nécessaires quand les mêmes fonctionnalités sont implantées dans un programme externe communiquant en mode client/serveur avec la base de données.

PLSQL est un langage de requête procédurale. C'est une extension de SQL qui permet de d'introduire des blocs de programmation procédurale dans le code d'une requête. Il permet aussi d'envoyer un bloc de requêtes SQL au serveur en une seule fois. A travers les procédures, les fonctions, les packages, les déclencheurs de la base de données, etc…, PLSQL joue aussi un rôle central entre le serveur Oracle et les différents outils de développement d'Oracle. Un bloc PL/SQL est intégralement envoyé au moteur PL/SQL, qui traite chaque instruction PL/SQL et sous-traite les instructions purement SQL au moteur SQL, afin de réduire le trafic réseau.

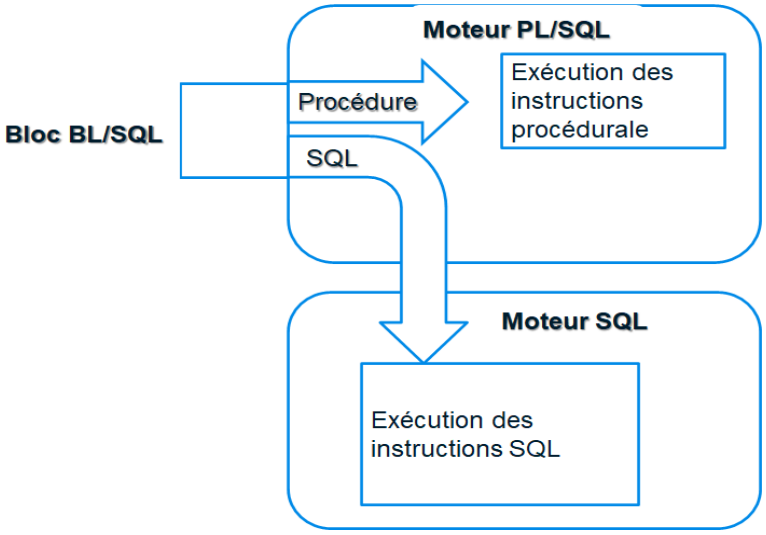


Figure-1 : Environnement : PLSQL

La syntaxe du PLSQL est composée de celle de SQL plus :

- Variables et constantes
- Traitements conditionnels
- Traitements itératifs
- Curseurs en PL/SQL
- Les enregistrements et les tables PL/SQL
- Procédures
- Fonctions
- Les packages en Oracle
- Le traitement des exceptions (erreurs)

Chaque bloc de PLSQL est composé principalement de trois parties qui sont : la partie déclarative qui commence par la commande « DECLARE », le corps du bloc qui commence par « GEGIN » et une partie de gestion des exceptions qui commence par « EXEPTION ». Le bloc se termine par « END ».

```
DECLARE
        Variables;
        Constantes;
        Curseurs;
        Tables PL/SQL;
        Enregistrements PL/SQL;
        Exceptions;
BEGIN
        Instructions SQL et PL/SQL
EXCEPTION
```

Traitement des exceptions (gestion des erreurs)
END

Chaque instruction se termine par ';'
- Les parties 'DECLARE' et 'EXCEPTION' sont facultatives
- On peut inclure des commentaires dans un bloc PL/SQL en utilisant la syntaxe suivante :
-- commentaires sur une seule ligne
/* .....*/ commentaires sur plusieurs lignes.

II- Les variables scalaires

C'est un moyen pour stocker en mémoire une valeur d'un certain type. Une variable est caractérisée par un nom, une adresse mémoire et une valeur.

Le nom des variables ne doit pas être le même que celui d'une colonne ou d'une table de la base de données. L'affectation d'une valeur à une variable se fait par : ':=', DEFAULT.
Exemple : v_PU := 25.25 ;
v_Ville DEFAULT 'Casablanca' ;

Dans PLSQL on peut utiliser trois types de variables :

II-2- Variables de type Oracle :

- Char, Varchar2, Number, Date, Long

II-3- Variables de type booléens :

- BOOLEAN

NB. : On utilise sous SQL*Plus la commande : **SET SERVEROUTPUT ON** pour pouvoir afficher les résultats de la commande Oracle (Package) :
DBMS_OUTPUT.PUT_LINE(chaîne).

Le **DBMS_OUTPUT** package permet d'envoyer des messages à partir de procédures stockées, de packages et de déclencheurs. Le package est particulièrement utile pour afficher les informations de débogage PL / SQL.

Exemple 1:
Soit la relation suivante :

Produit(Ref, Nom, PU, Stk)

Ref : entier 2 chiffres
Nom : chaine de 10 char
PU : réel de 4 chiffres partie entière et 2 chiffres partie fractionnaire
Stk : entier 3 chiffres

On veut déclarer des variables de même type que les attributs de la table Produits leur affecter des valeurs puis les afficher :

```
SET SERVEROUTPUT ON
DECLARE
     v_Ref Number(2, 0) := 12;
     v_Nom varchar2(10) := 'Ordinateur';
     v_PU Number(6, 2) := 235,12;
     v_Stk Number(3, 0) := 254;
BEGIN
     DBMS_OUTPUT.PUT_LINE('Ref Produit : '||v_Ref);
     DBMS_OUTPUT.PUT_LINE('Nom Produit : '||v_Nom);
     DBMS_OUTPUT.PUT_LINE('Prix : '||v_PU);
     DBMS_OUTPUT.PUT_LINE('Stock : '||v_Stk);
END;
```

Remarque :
Le paramètre **SERVEROUTPUT** contrôle si SQL * Plus imprime la sortie générée par le package DBMS_OUTPUT à partir des procédures PL / SQL.
le **DBMS_OUTPUT** package permet d'envoyer des messages à partir de procédures stockées, de packages et de déclencheurs. Le package est particulièrement utile pour afficher les informations de débogage PL / SQL.

Exemple 2 :
On veut maintenant insérer les valeurs de ces 4 variables dans la table produit.

```
DECLARE
     v_Réf Number(2, 0) := 16;
     v_Nom varchar2(10) := 'Ordinateur';
     v_PU Number(6, 2) := 356.12;
     v_Stk Number(3, 0) := 254;
BEGIN
     INSERT into PRODUIT(réf, Nom, PU, Stk)
     VALUES (v_Réf, v_Nom, v_PU, v_Stk);
END;
```

Exemple 3 :
On veut maintenant affecter à ces 4 variables les valeurs du produit dont la référence est 213.

```
SET SERVEROUTPUT ON
DECLARE
     v_Réf Number(2, 0);
     v_Nom varchar2(10);
     v_PU Number(6, 2);
     v_Stk Number(3, 0);
```

```
BEGIN
     SELECT Réf, Nom, PU, Stk
     INTO v_Réf, v_Nom, v_PU, v_Stk
     FROM PRODUIT
     WHERE "RÉF"=16;
     DBMS_OUTPUT.PUT_LINE('Ref Produit : '||v_Réf);
     DBMS_OUTPUT.PUT_LINE('Nom Produit : '||v_Nom);
     DBMS_OUTPUT.PUT_LINE('Prix : '||v_PU);
     DBMS_OUTPUT.PUT_LINE('Stock : '||v_Stk);
END;
```

II-3- Variables faisant référence au dictionnaire de données.

On peut définir de nouveau type de variables qui font référence à la structure d'une table de la BDD. On utilise donc la close « %TYPE ».

II-2-1- Variables de même type qu'un attribut d'une table de BD :

On utilise la syntaxe suivante :

```
DECLARE
Variable Nomtable.Attribut%TYPE;
```

Exemple1: reprendre l'exemple-3 on utilisant des variables de type attributs.

```
SET SERVEROUTPUT ON
DECLARE
v_Réf PRODUIT.RÉF%TYPE;
v_Nom Produit.Nom%TYPE;
v_PU Produit.PU%TYPE;
v_Stk Produit.Stk%TYPE;
BEGIN
     SELECT Réf, Nom, PU, Stk INTO v_Réf, v_Nom, v_PU, v_Stk
    FROM PRODUIT
     WHERE "RÉF"=16;
DBMS_OUTPUT.PUT_LINE('Ref Produit : '||v_Réf);
DBMS_OUTPUT.PUT_LINE('Nom Produit : '||v_Nom);
DBMS_OUTPUT.PUT_LINE('Prix : '||v_PU);
DBMS_OUTPUT.PUT_LINE('Stock : '||v_Stk);
END;
```

II-2-2- Variables de même type qu'un n-uplet d'une table de BD :

On utilise la syntaxe suivante :

```
DECLARE
Variable Nomtable%ROWTYPE;
```

Pour accéder à un attribut on écrit :

```
Variable.NomAttribut ;
```

Exemple2: reprendre l'exemple-3 on utilisant des variables de type ligne (Row).

```
SET SERVEROUTPUT ON
DECLARE
v_ligne PRODUIT%ROWTYPE;
BEGIN
     SELECT Réf, Nom, PU, Stk INTO v_ligne
    FROM PRODUIT
     WHERE "RÉF"=16;
     DBMS_OUTPUT.PUT_LINE(v_ligne.Réf||''||v_ligne.Nom||
     ''||v_ligne.PU||''||v_ligne.Stk);
END;
```


III- Les variables composées (Les enregistrements et Les Tables)

Pour Stocker en mémoire des colonnes ou des lignes de données Oracle et de les faire passer en tant que paramètres de fonctions ou de procédures PL/SQL, et afin faciliter le déplacement des collections de données dans et hors des tables de base de données ou entre les applications côté client et les sous-programmes stockés, on doit utiliser les tables et ou les enregistrements.

III-1 Les Collections ou les Tables (Table)

Une table PL / SQL est une collection ordonnée d'éléments du même type. PLSQL d'oracle propose trois types de tables :

Les tables d'indexation : ce sont des tables de taille dynamique et qui peuvent être indexés par des valeurs numériques ou alphanumériques.

Les tables imbriquées : Ce sont aussi des tables de taille dynamique et qui sont indexés par des variables numériques. Ils peuvent être stocké dans une colonne de la BDD.

Les tables Varray : ou les tables à taille variable indexés par des variables numériques, dont le nombre d'éléments maximum est fixé dès leur déclaration et Ils peuvent être stocké dans une colonne de la BDD.

a- *Les tables d'indexation*

La taille de ces tables est dynamique et elles ne peuvent pas être stockées dans une colonne de la BDD. Les éléments du tableau peuvent avoir le type des champs de la BDD et l'accès à un élément de la table s'effectue grâce à un numéro d'index unique qui détermine sa position dans la collection ordonnée. L'indexation peut donc se faire par des index significatifs. Pas obligé d'indexer par des numéros consécutifs. Cet index est déclaré de type numérique ou chaine de caractère (BINARY_INTEGER pour des indexes à valeurs entières signées). Les tables d'indexation ne peuvent avoir qu'une seule dimension. Mais en créant des collections de collections on peut obtenir des tableaux à plusieurs dimensions.

L'accès à un élément de la table s'effectue grâce à un indice. Cet indice est déclaré de type INTEGER ou STRING. La déclaration se fait en deux étapes :

La déclaration se fait en deux étapes :

- Déclaration du type de l'élément de la table :

```
TYPE nom-du-type IS TABLE OF type argument
INDEX BY BINARY_INTEGER;
```

- Déclaration de la variable de type table :

```
nom-variablenom-du-type;
```

Exemple1 :

```
set Serveroutput on;
declare
type tab is table of float
Index by Varchar2(20);
PU tab;
begin
PU('Produit1'):=12.5;
PU('Produit2'):=13.9;
dbms_Output.Put_Line(PU('Produit1'));
end;
```

le programme affichera 12.5

Exemple2 :

```
set Serveroutput on;
declare
```

```
type tab is table of Varchar2(20)
Index by Varchar2(4);
Emp tab;
begin
Emp('LC31'):='BERRADA Ahmed';
Emp('BE37'):='ALAOUI Khalid';
Emp('AX98'):='KARIMI Lamiaa';
Emp('BY45'):='YASSINE Anas';
dbms_Output.Put_Line(Emp('BE37'));
end;
```

Le programme affichera `'ALAOUI Khalid'`.

Exemple3 :
Stocker en mémoire dans un tableau les noms des sociétés des clients dont le codeclient est « BONAP » et « COMMI ».

```
SET SERVEROUTPUT ON
DECLARE
     TYPE TableCode is TABLE of VARCHAR2(100)
     index by BINARY_INTEGER;
     T TableCode;
     i number(1,0) :=0;
BEGIN
     select "SOCIÉTÉ" into T(i)
     from CLIENTS
     where CODECLIENT = 'BONAP';
     DBMS_OUTPUT.PUT_LINE(T(i));
     i:=i+1;
     select "SOCIÉTÉ" into T(i)
     from CLIENTS
     where CODECLIENT = 'COMMI';
     DBMS_OUTPUT.PUT_LINE(T(i));
END;
```

b- Les tables imbriquées.

Ce sont aussi des tables à taille dynamique. A la différence des tables d'indexation, les tables imbriquées ne peuvent être indexé que par des entiers signés (entre 1 et 2^{31}) et peuvent être stockées dans une colonne d'une table de la BDD oracle. On peut créer un nouveau type de données « table » et affecter ce dernier à l'un des attributs d'une table de la BDD oracle. Les

tables imbriquées sont donc utilisées dans la Gestion d'attribut multivalué par imbrication de collection de scalaires (BDD Relationnel-Objet).

Contrairement aux tables d'indexation, les tables imbriquées doivent être initialisées et doivent être étendues avec l'indicateur **« Extend »** avant chaque ajout d'un nouvel élément.

La déclaration d'une table imbriquée se fait de la même manière de la table d'indexation **mais sans déclarer le type d'indexe** (index by ….)

-- Déclaration du type de l'élément de la table :

```
TYPE nom_Type IS TABLE OF type_valeur;
```

--Déclaration et initialisation de la variable de type table :

```
nom-variable nom_Type : = nom_Type();
```

-- Extension du tableau avant chaque ajout d'élément

```
nom-variable.Extend
```

Exemple1 : Nous voulons stockées les clients de la BDD dans un tableau imbriqué

```
set Serveroutput on;
declare
cursor cur is select * from clients;
type tab is table of Clients%rowtype;
T tab:=tab();
i integer:=1;
begin
T.extend;
for n in cur loop
T(i):=n;
i:=i+1;
T.extend;
end loop;
Dbms_Output.Put_Line(T(1).idclient||'  '||T(1).societe||'
'||T(1).ville);
Dbms_Output.Put_Line(T(2).idclient||'  '||T(2).societe||'
'||T(2).ville);
END;
```

Exemple2 : Nous voulons saisir les données concernant des personnes avec pour chacun deux plusieurs numéros de téléphone.

```
-- création d'un type de données "table"
create or replace type T_Nesteb is table of varchar2(10);
-- création d'une table de BDD avec un attribut de type
"table"
create table personne (id integer Primary Key, Nom
Varchar2(100), Tel T_Nesteb)
nested table Tel store as personne_TEL;
-- insertion de données dans la table principale et dans la
table imbriquée
insert into Personne values (1, 'Amine',
T_Nesteb('0525889977', '0661615547', '0671887745'));
insert into Personne values (2, 'Alaa', T_Nesteb('0525889977',
'0661615547', '0671887745'));
insert into Personne values (3, 'Zineb',
T_Nesteb('0525889977', '0661615547', '0671887745'));
```

ID	NOM	TEL
1	Amine BERRADA	BDDVENTE.T_NESTEB('0525889977', '0661615547', '0671887745')
2	Alaa ALAMI	BDDVENTE.T_NESTEB('0525889977', '0661615547', '0671887745')
3	Zineb KARIMI	BDDVENTE.T_NESTEB('0525889977', '0661615547', '0671887745')

III-1- Les Enregistrements (RECORD)

Permet de stocker en mémoire sous le même identificateur un ensemble de valeur de différents types

- Par référence à une structure de table ou de curseur en utilisant « ROWTYPE ». Les curseurs seront traités dans les paragraphes suivants.

    - Déclaration du type enregistrement

      ```
      TYPE nom-du-type IS RECORD (
      nom-variable nom-table%ROWTYPE;...);
      ```

    - Déclaration de la variable de type enregistrement

      ```
      nom-variable nom-du-type;
      ```

- Par énumération des rubriques (champs) qui la composent.

    - Déclaration du type enregistrement

      ```
      TYPE nom-du-type IS RECORD (
      nom-attribut1 type-attribut1,
      ```

```
nom-attribut2 type-attribut2,
...);
```

- Déclaration de la variable de type enregistrement

```
nom-variable nom-du-type;
```

Exemple :

a- Stocker en mémoire dans un enregistrement le n-uplet de la table clients dont le codeclient='BONAP'

```
SET SERVEROUTPUT ON
DECLARE
     TYPE Str is RECORD
     (CL Clients%rowtype);
     P Str;
BEGIN
     select * into P.CL
     from CLIENTS
     where Codeclient='BONAP';
     DBMS_OUTPUT.PUT_LINE(P.CL.CODECLIENT||'
     '||P.CL.SOCIéTé||' '||P.CL.FONCTION);
END;
```

b- Stocker en mémoire dans un enregistrement seulement les valeurs des attributs CODECLIENT, SOCETE, FONCTION de la table clients dont le codeclient='BONNAP'

```
SET SERVEROUTPUT ON
DECLARE
TYPE Str is RECORD
(Code Clients.CODECLIENT%TYPE,
  Nom CLIENTS."SOCIÉTÉ"%type,
  Fonc CLIENTS.FONCTION%type);
P Str;
BEGIN
select CODECLIENT, société, Fonction into P
from CLIENTS
where Codeclient='BONAP';
DBMS_OUTPUT.PUT_LINE(P.code||' '||P.Nom||'
'||P.Fonc);
END;
```

IV- Les Structures de contrôle

Permettent de contrôler l'exécution linéaire des instructions dans un programme informatique.

III-1- Les Structures de contrôle alternatives (conditionnelles)
Permettent donner plusieurs issus d'exécution d'un programme.
IF condition1 **THEN**
traitement1;
ELSIF condition2 **THEN**
traitement2;
[**ELSE**
traitement3;]
END IF;

Exemple-1 :

Mettre en mémoire la référence, la désignation et le prix unitaire du produit dont la référence est 10. Puis afficher si ce produit est chair ou pas.

- Si le prix unitaire >100 le produit est chair
- Si le prix unitaire est compris 100 et 200 le produit est un peu chair
- Si le prix unitaire <100 le produit n'est pas chair

```
SET SERVEROUTPUT ON
DECLARE
     R PRODUITS."RÉFPRODUIT"%type;
     D Produits.designation%type;
     PU Produits.prixunitaire%type;
BEGIN
  select Réfproduit, Designation, Prixunitaire
  into R, D, PU
  from produits
  where "RÉFPRODUIT"=10;
  DBMS_OUTPUT.PUT_LINE('Ref'||' '||R);
  DBMS_OUTPUT.PUT_LINE('Nom'||' '||D);
   DBMS_OUTPUT.PUT_LINE('PU'||' '||PU);
  if PU<100 then
    DBMS_OUTPUT.PUT_LINE('produit pas chair');
  elsif PU<=200 then
   DBMS_OUTPUT.PUT_LINE('produit pas trop chair');
  else
  DBMS_OUTPUT.PUT_LINE('produit est chair');
  end IF;
END;
```

Exemple-2 :

Mettre en mémoire le codeclient, la société et le chiffre d'affaire CA du client dont le id égale à 10. Puis accorder lui la commission suivante.

- Si le CA < 1000 : 20% du CA
- Si le CA est compris 1000 et 2000 : 30% du CA
- Si le CA > 2000 : 40% du CA

```
SET SERVEROUTPUT ON
declare
type str is RECORD (
code clients.idclient%type,
nom clients.societe%type,
CA float,
CO float);
S str;
begin
select clients.idclient, societe,
sum(prixunitaire*quantite*(1-remise/100))
into S.code, S.nom, S.CA
from clients, commandes, lignecommandes, produits
where clients.idclient=commandes.idclient
and commandes.idcommande=lignecommandes.idcommande
and lignecommandes.idproduit=produits.idproduit
and clients.idclient=10
group by clients.idclient, societe;
if S.CA<1000 then
    S.CO:=0.2*S.CA;
elsif S.CA<2000 then
    S.CO:=0.3*S.CA;
else
    S.CO:=0.4*S.CA;
end if;
DBMS_OUTPUT.PUT_LINE(S.code||'  '||S.nom||'
'||S.CA||'  '||S.CO);
end;
```

III-2- Les Structures de contrôle itératives

Il existe trois types de boucles :

1. La boucle LOOP (de base)
2. La boucle FOR
3. La boucle WHILE

III-2-1 la boucle LOOP

Syntaxe :

LOOP
    Expression1;
    Expression2;
    .
    .
    .
    EXIT [WHEN condition];
END LOOP;

Exemple :

Créer dans une base de données un tableau pour stocker une table de multiplication d'un nombre dont la valeur va être saisie par l'utilisateur.

```
CREATE TABLE Mult(Op char(6), res number(3,0));
DECLARE
  Compteur NUMBER(2,0) := 0;
  x number(2,0);
  y number(2,0);
BEGIN
  y:=&x;
  LOOP
  Compteur := compteur + 1;
   INSERT INTO Mult
   VALUES(TO_CHAR(compteur)||'*'||TO_CHAR(y),
   compteur*y);
  EXIT WHEN compteur = 10;
  END LOOP;
END;

SELECT * FROM Mult;

ROLLBACK;
```

NB:

- « **ROLLBACK** » sert à annuler l'opération de l'insertion si on n'a pas encore validé avec « **COMMIT** ».
- Précéder une variable par « & » signifie que la valeur de cette variable sera saisie au cours de l'exécution du programme.

III-2-2. la boucle FOR

Exécution d'un traitement un certain nombre de fois, le nombre étant connu à l'avance.

Syntaxe :

BEGIN
…
FOR indice IN [REVERSE] exp1..exp2
LOOP
Instructions;
END LOOP;
END;

N.B. :

- Inutile de déclarer la variable 'indice'
- Indice varie de exp1 à exp2 avec un pas de 1 (exp1<exp2)
- Si 'REVERSE' est précisé alors 'indice' varie de exp1 à exp2 avec un pas de –1 (exp1>exp2)

Exemple :
Refaire l'exo de la table de multiplication en utilisant « for »

```
CREATE TABLE Mult(Op char(5), res Number(3,0));
DECLARE
      x number(2,0);
      y number(2,0);
BEGIN
      y:=&x;
      for i in 1..10
        LOOP
          INSERT INTO Mult VALUES(
          TO_CHAR(i)||'*'||TO_CHAR(y),i*y);
      END LOOP;
END;
```

III-2-3. la boucle WHILE

Exécution d'un traitement tant qu'une condition est vraie.

Syntaxe :

WHILE condition
LOOP
Instructions;
END LOOP;

Exemple :
Refaire l'exo de la table de multiplication en utilisant « while »

```
CREATE TABLE Mult(Op char(5), res Number(3,0));
DECLARE
      Compteur NUMBER := 0;
      x number(2,0);
      y number(2,0);
BEGIN
      y:=&x;
      while(compteur <=10)
            LOOP
              Compteur := compteur + 1;
              INSERT INTO Mult VALUES(
              TO_CHAR(compteur)||'*'||TO_CHAR(y),
              compteur*y);
            END LOOP;
END;
```


V- Les curseurs

C'est un outil qui permet de parcourir une par une les lignes récupérer par une requête de sélection (SELECT). Il contient les lignes (une ou plusieurs) renvoyées par une instruction SQL. Il existe deux types de curseurs -

- Curseurs implicites
- Curseurs explicites

IV-1 Curseurs implicites

Ils sont automatiquement créés par Oracle chaque fois qu'une instruction SQL est exécutée en l'absence de curseur explicit. Les programmeurs ne peuvent pas contrôler les curseurs implicites et les informations qu'ils contiennent.

Chaque fois qu'une instruction DML (INSERT, UPDATE et DELETE) est émise, un curseur implicite est associé à cette instruction. Pour les opérations INSERT, le curseur contient les données qui doivent être insérées. Pour les opérations UPDATE et DELETE, le curseur identifie les lignes qui seraient affectées.
Dans PL / SQL, on peut faire référence au curseur implicite le plus récent en tant que curseur SQL, qui a toujours des attributs tels que %FOUND, %ISOPEN, %NOTFOUND et %ROWCOUNT.

Exemple : Créer un bloc PLSQL qui permet de faire la mise à jour des prix des produits du fournisseur idfour=10, Cette mise à jour est une augmentation de 20%. Le programme doit aussi afficher le nbre de produit qui ont été mise à jour ou bien le message « pas de produit » s'il n'existe pas de produit du fournisseur 10.

```
set SERVEROUTPUT ON
declare
nbre int;
begin
update produits set prixunitaire=prixunitaire*(1+0.20) where idfour=10;
if sql%notfound then
        dbms_output.put_line('pas d'employe selectione');
else
        nbre:=sql%rowcount;
        dbms_output.put_line(nbre);
end if;
end;
```


IV-2 Curseurs explicites

Les curseurs explicites sont des curseurs définis par le programmeur pour obtenir plus de contrôle. Un curseur explicite doit être défini dans la section déclaration du bloc PL / SQL. Il est créé sur une instruction « SELECT » qui renvoie plus d'une ligne. Le fonctionnement d'un curseur explicite se compose de trois étapes.

1. Déclaration du curseur avec sa requête SELECT
2. L'ouverture du curseur
3. Récupération des lignes une à une, en commençant par la première
4. Fermeture du curseur

a- Déclaration du curseur

Elle se fait dans la partie DECLARE du programme PLSQL.

Syntaxe :

```
DECLARE
CURSOR nom_Curseur IS
SELECT Champ1, Champ2, ….
        FROM Table
   WHERE < Condition >        } Requête SQL
      …
BEGIN
   ….
```

END;

b- Ouverture du curseur :

L'ouverture du curseur lance l'exécution de l'ordre SELECT associé au curseur. L'ouverture se fait dans la section BEGIN du bloc.

Syntaxe :

DECLARE
CURSOR nom_Curseur **IS**
$SELECT\ Champ1, Champ2, \ldots.$
$FROM\ Table$ } Requête SQL
$WHERE < Condition >$
…
BEGIN
OPEN nom_Curseur;
…..;
END;

c- Traitement de lignes :

Après l'exécution du SELECT, les lignes ramenées sont traitées une par une, la valeur de chaque colonne du SELECT doit être stockée en mémoire en utilisant une variable réceptrice.

Syntaxe :

DECLARE
CURSOR nom_Curseur **IS**
$SELECT\ Champ1, Champ2, \ldots.$
$FROM\ Table$ } Requête SQL
$WHERE < Condition >$
…
BEGIN
OPEN nom_Curseur;
FETCH nom_Curseur **INTO** liste_variables;
….
CLOSE nom Curseur
END;

Remarque:
On peut utiliser une écriture implicite plus simple :

DECLARE
CURSOR nom_Curseur **IS**

$$\left.\begin{array}{l} SELECT\ Champ1, Champ2, \ldots. \\ \quad FROM\ Table \\ WHERE < Condition > \end{array}\right\} Requête\ SQL$$

…

BEGIN

FOR n in nom_Curseur **loop**

…..

END LOOP;

END;

Exemple-1 :

Stocker en mémoire dans un tableau d'enregistrement puis afficher les valeurs des attributs CODECLIENT, SOCIETE, FONCTION.

```
SET SERVEROUTPUT ON
DECLARE
TYPE Str is RECORD
(Code Clients.CODECLIENT%TYPE,
  Nom CLIENTS.societe%type,
  Fonc CLIENTS.FONCTION%type);
TYPE TableClient is TABLE of Str
index by BINARY_INTEGER;
TP TableClient;
i integer:=0;
CURSOR cur_cl IS
SELECT CODECLIENT, societe, FONCTION
FROM CLIENTS;
BEGIN
OPEN cur_cl;
FETCH cur_cl INTO TP(i);
while cur_cl%found loop
    i:=i+1;
    FETCH cur_cl INTO TP(i);
end loop;
for k in 0..i-1 loop
    DBMS_OUTPUT.PUT_LINE(k||'  '||TP(k).CODE||'
'||TP(k).Nom||'  '||TP(k).Fonc);
end loop;
END;
```

Exemple-2 :
Stocker en mémoire dans un tableau PL/SQL puis afficher les valeurs du champ « Société » de la table CLIENTS.

```
SET SERVEROUTPUT ON
DECLARE
TYPE TableCode is TABLE of clients.societe%type
index by BINARY_INTEGER;
T TableCode;
i integer:=0;
CURSOR cur_cl IS SELECT societe FROM
 CLIENTS;
BEGIN
OPEN cur_cl;
FETCH cur_cl INTO T(i);
while cur_cl%found loop
    i:=i+1;
    FETCH cur_cl INTO T(i);
end loop;
for k in 0..i-1 loop
    DBMS_OUTPUT.PUT_LINE(k||'  '||T(k));
end loop;
END;
```


Exemple-3 :
Stocker maintenant en mémoire dans un tableau d'enregistrement PL/SQL puis afficher toutes les lignes du tableau CLIENTS.

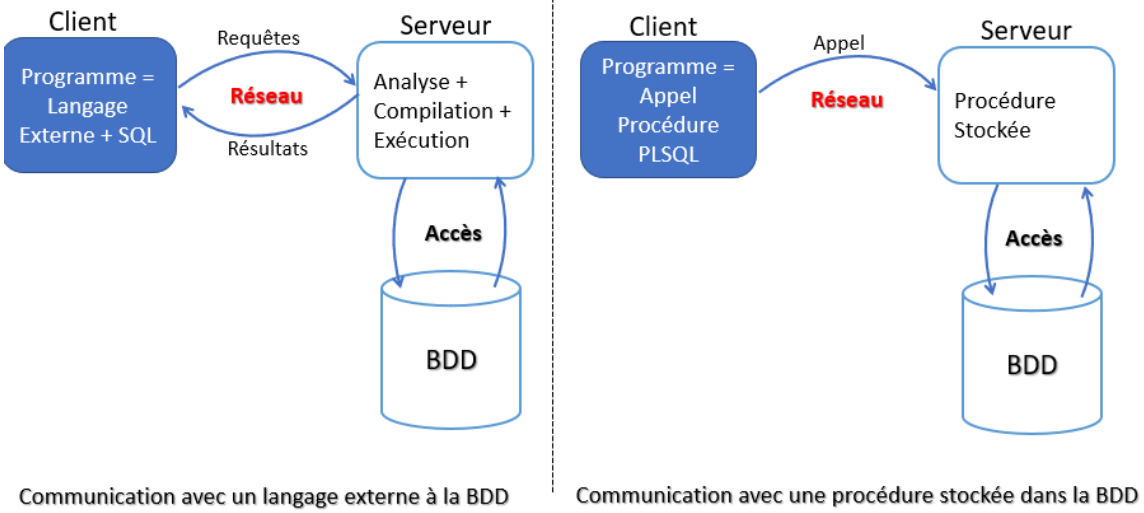
```
SET SERVEROUTPUT ON
DECLARE
TYPE Str is RECORD
(E Clients%rowtype);
TYPE TableClient is TABLE of Str
index by BINARY_INTEGER;
TP TableClient;
i integer:=0;
CURSOR cur_cl IS
SELECT *
FROM CLIENTS;
BEGIN
OPEN cur_cl;
FETCH cur_cl INTO TP(i).E;
while cur_cl%found  loop
    i:=i+1;
    FETCH cur_cl INTO TP(i).E;
end loop;
for k in 0..i-1 loop
```

```
    DBMS_OUTPUT.PUT_LINE(k||'
'||TP(k).E.CODECLIENT||'  '||TP(k).E.societe||'
'||TP(k).E.Fonction);
end loop;
END;
```

VI- Les Procédures et les Fonctions

V-1 Les Procédures Stockées

Une procédure est un bloc PL/SQL nommé qui exécute une ou plusieurs actions et requêtes.
Elle peut être stockée dans la base de données, comme tout autre objet, et peut être appelée et exécuter plusieurs fois en cas de besoin.
Une procédure stockée s'exécute au sein du SGBD, ce qui évite les échanges réseaux qui sont nécessaires quand les mêmes fonctionnalités sont implantées dans un programme externe communiquant en mode client/serveur avec la base de données.
Dans le cas des programmes externe, en C ou Java par exemple, l'application doit tout d'abord se connecter au serveur du SGBD. Le programme s'exécute alors en communiquant avec le serveur pour exécuter les requêtes et récupérer les résultats. Dans cette architecture, chaque demande d'exécution d'un ordre SQL implique une transmission sur le réseau, du programme vers le client, suivie d'une analyse de la requête par le serveur, de sa compilation et de son exécution.
Dans le cas des procédures stockées (PS) au niveau du serveur on a la possibilité de regrouper du côté de ce dernier l'ensemble des requêtes SQL et le traitement des données récupérées. La procédure est compilée une fois par le SGBD, au moment de sa création, ce qui permet de l'exécuter rapidement au moment de l'appel. De plus les échanges réseaux ne sont plus nécessaires puisque la logique de l'application est étroitement intégrée aux requêtes SQL. Le rôle du programme externe se limite alors à se connecter au serveur et à demander l'exécution de la procédure, en lui passant au besoin les paramètres nécessaires.



Communication avec un langage externe à la BDD

Communication avec une procédure stockée dans la BDD

Syntaxe :
CREATE [OR REPLACE] PROCEDURE nom_proc
(param1 [type][mode] type_donnée,
param2 [type][mode] type_donnée,
….)
IS
Bloc PL/SQL

Les **types** de paramètres ainsi que les **modes** de passages seront expliqués dans le paragraphe V-3

Exemple1 : créer une procédure qui permet d'afficher le nombre de commande réalisé pendant une année qui sera transmise en paramétrer.

```
create or replace PROCEDURE CAANN (ANN IN NUMBER ) IS
x NUMBER;
BEGIN
select count(idCommande) into x
from COMMANDES
GROUP by extract(year from datecommande)
having extract(year from datecommande)=ANN;
DBMS_OUTPUT.PUT_LINE(x);
exception
when no_data_found then
DBMS_OUTPUT.PUT_LINE('aucune ventes en '||ANN);
END;
```

Pour tester la procedure:

```
BEGIN
CAANN(2007);
end;
```

On peut écrire la requête sans utiliser le group et Having. Dans ce cas on a pas besoin de gérer une exception puisque la requête ne va pas générer une erreur.

```
create or replace PROCEDURE CAANN (ANN IN NUMBER ) IS
x NUMBER;
BEGIN
select count(idCommande) into x
from COMMANDES
where extract(year from datecommande)=ANN;
DBMS_OUTPUT.PUT_LINE(x);
END;
```

Exemple2 : créer une procédure qui permet d'insérer une ligne dans une table : PRODUIT(REFPRODUIT, NOMPRODUIT, PRIX, STOCK).

```
create or replace PROCEDURE testinser
(
x PRODUIT.REFPRODUIT%type,
y PRODUIT.NOMPRODUIT%type,
z PRODUIT.PRIX%type,
St PRODUIT.STOCK%type
)
IS
     BEGIN
       insert into Produit(REFPRODUIT, NOMPRODUIT, PRIX, STOCK)
     values (x, y, z, St);
END testinser;
```

Pour insérer :

```
BEGIN
 TESTINSER('az3', 'nom3', 23.43, 12);
end;
```


V-2 Les fonctions Stockées

Ce sont aussi des sous programmes PL/SQL stockés dans la BDD comme un objet nommé. Mais la particularité des fonctions c'est qu'elle revoie une valeur (RETURN).

Exemple : créer une fonction qui permet de renvoyer le nombre de commande réalisé pendant une année qui sera transmise en paramétrer.

```
create or replace FUNCTION FCAANN(ANN IN NUMBER) RETURN NUMBER
AS
x NUMBER;
BEGIN
select count(idCOMMANDE) into x
from COMMANDES
GROUP by extract(year from datecommande)
having extract(year from datecommande)=ANN;
RETURN x;
exception
when no_data_found then
RETURN 0;
END;
```

Pour tester la function:

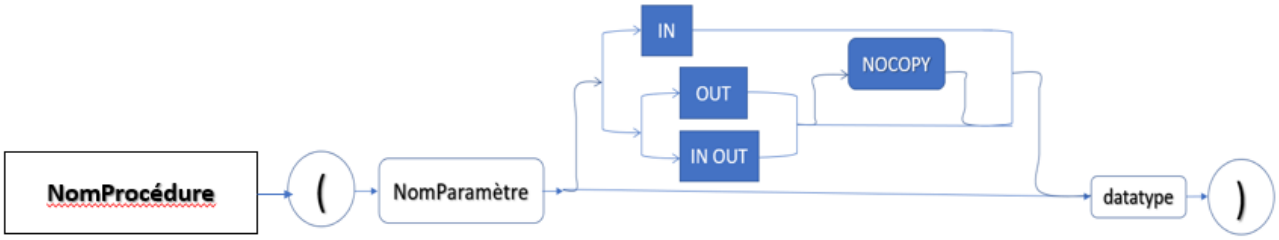
```
BEGIN
DBMS_OUTPUT.PUT_LINE(fCAANN(2007));
end;
```

ou bien :

```
create or replace FUNCTION FCAANN(ANN IN NUMBER) RETURN NUMBER
AS
x NUMBER;
BEGIN
select count(idCOMMANDE) into x
from COMMANDES
where extract(year from datecommande)=ANN;
RETURN x;
END;
```

V-3 Les Paramètres dans les fonctions et les procédures.

Les fonctions et les procédures sont paramétrable afin de faciliter leur utilisation. Trois types de paramètres peuvent être utilisés dans PLSQL : « IN », « OUT » ou « IN OUT » respectivement les paramètres d'entrés de sorties ou d'entrées-Sorties. Dans le cas des paramètres OUT et IN OUT le passage peut être par valeur ou par référence. Les passages par référence se fait en utilisant l'indicateur « NOCOPY ».



Déclaration des Paramètres

a- Type de transmission (IN, OUT, IN OUT)

IN : (valeur par défaut) indique que le paramètre transmis par le programme appelant n'est pas modifiable par la procédure.

OUT : indique que le paramètre est modifiable par la procédure.

IN OUT : indique que le paramètre est transmis par le programme appelant et modifier par la procédure.

Remarque : Si aucun type de transmission n'est défini le paramétré aura par défaut une transmission « IN »

b- Mode de transmission (par valeur ou par référence)

Afin d'améliorer les performances des paramètres de sorties des d'entrés-Sortie (OUT et IN OUT), PLSQL propose deux modes de transmission : transmission par valeur et transmission par référence.

- **Passage par valeur :** c'est le mode par défaut. Une fois la valeur est transmise, par le programme appelant, un tampon temporaire est créé (paramètre formel). La donnée de la variable de paramètre réel est copiée vers ce tampon. La procédure travaille sur ce tampon temporaire pendant son exécution. Une fois la procédure terminée, le contenu du tampon temporaire est recopié dans la variable de paramètre. C'est cette opération de recopie du tampon vers le paramètre réel qui nous permet de récupérer la donnée modifiée par une procédure même si on fait un passage par valeur. Mais **En cas d'exception, l'opération de recopie n'a pas lieu.**
- **Passage par référence :** l'utilisation de l'indicateur **NOCOPY** indique au compilateur d'utiliser le passage par référence. Dans ce cas aucun tampon temporaire n'est nécessaire et aucune opération de copie en avant et en après l'exécution de la procédure ne se produit. Puisque c'est l'adresse mémoire de la donnée est transmise toute modification des valeurs de paramètre est écrite directement dans la variable de paramètre (paramètre réel).

Dans le cas d'un traitement d'une quantité de données normale, nous n'allons pas remarquer la différence entre les deux méthodes, mais une fois que nous commençons à manipuler des données volumineux ou complexes, la différence entre les deux modes de passage peut être assez considérable. La présence du tampon temporaire signifie passer par valeur nécessite deux fois la mémoire pour chaque paramètre OUT et IN OUT, ce qui peut être un problème lors de l'utilisation de grands paramètres. De plus, le temps nécessaire pour copier les données dans le tampon temporaire et les retourner dans la variable de paramètre peut être assez considérable.

Exemple : Ecrire une procédure qui permet de calculer la réduction accordée à un client donné. Cette réduction est accordée en fonction du chiffre d'affaire (CA). Elle est de 0% du CA si ce dernier est inferieur à 1000. De 20% si 1000<CA<=10000 et 30% si CA>10000. Le CA et la réduction RD seront des paramètres de la procédure. Appeler cette procédure dans une programme PLSQL.

```
create or replace procedure
calculCom(CA IN float, Rd OUT nocopy float) as
begin
if CA<1000 then
    Rd:=0*CA;
elsif CA<10000 then
```

```
        Rd:=0.2*CA;
    else
        Rd:=0.3*CA;
end if;
end;

set Serveroutput On;
declare
code clients.codeclient%type;
Nom clients.societe%type;
XCA float;
XRD float;
begin
select codeclient, societe, sum(prixunitaire*quantite*(1-
remise/100)) into code, Nom, XCA
from clients, commandes, Lignecommandes,Produits
where clients.idclient=commandes.idclient
and Commandes.Idcommande=Lignecommandes.Idcommande
and Lignecommandes.Idproduit=Produits.Idproduit
group by clients.idclient, codeclient, societe
Having clients.idclient=1;
Calculcom(XCA, XRD);
Dbms_Output.Put_Line(XRD);
end;
```


c- Transmission des enregistrements (reccords) : Curseur de Référence

VII- Les packages

C'est un ensemble de procédures, de fonctions, de variables, de constantes, de curseurs et d'exceptions stockés dans la base de données Oracle.
Un package est conçu en deux parties : une partie description du package et une partie corp du package.

- **Spécification :** C'est la partie déclaration de tous les composants du package : les procédures, les fonctions, les variables, etc….
- **Corps du package :** c'est la partie où on définit les procédures, les fonctions, les variables, les constantes, les curseurs et les exceptions du package.

Syntaxe :

CREATE [OR REPLACE] PACKAGE nom_package IS
    Section declaration;
END;
CREATE PACKAGE BODY nom_package IS
    Corps des procédures;
    Corps des fonctions;
    Déclaration des variables;
    Déclaration des constantes;
    Corps des curseurs;
    Déclaration des exceptions;
END;

Exemple1 :

Créer un package contenant :

1. Une procédure qui permet d'afficher le chiffre d'affaire par année
2. Une fonction qui permet de renvoyer le chiffre d'affaire réalisé pendant une année qui transmise en argument.
3. Un curseur qui permet de parcourir les ligne de la requête « select » qui calcule le CA par année.
4. Deux variables globales pour stocker en mémoire le CA et l'année.

Réponse :

```
CREATE OR REPLACE PACKAGE Pack IS
     Procedure CA_ANN;
     Function FCA_ANN(ANN NUMBER) return number;
     cursor CAANN is
     select Extract(year from datecommande),
     sum(Quantité*Prixunitaire)
     from commandes, détailscommandes, produits
     where commandes.Ncommande=détailscommandes.Ncommande
     AND détailscommandes.Réfproduit=PRODUITS."RÉFPRODUIT"
     group by Extract(year from datecommande);
     CA number;
     AN number;
END;
CREATE PACKAGE BODY Pack IS
Function FCA_ANN(ANN NUMBER) return number is
     begin
     select Extract(year from datecommande),
     sum(Quantité*Prixunitaire) into AN, CA
     from commandes, détailscommandes, produits
     where commandes.Ncommande=détailscommandes.Ncommande
```

```
        AND détailscommandes.Réfproduit=PRODUITS."RÉFPRODUIT"
        group by Extract(year from datecommande)
        having Extract(year from datecommande)=ANN;
        RETURN CA;
end FCA_ANN;
Procedure CA_ANN is
        begin
        open CAANN;
        fetch CAANN into AN, CA;
        while CAANN%found
        loop
        DBMS_OUTPUT.PUT_LINE(AN||' '||CA);
        fetch CAANN into AN, CA;
        end loop;
        close CAANN;
end CA_ANN;
END;
```

Pour exécuter :

```
set serveroutput on;
begin
DBMS_OUTPUT.PUT_LINE(PACK.FCA_ANN(2007));
PACK.CA_ANN;
end;
```


VII-1- Passage d'une table de données en paramètre d'une procédure stockées

Grace aux package on peut faire passer en paramètre d'une fonction ou d'une procédure un ensemble de données stockées en mémoire dans un tableau de valeur ou de record. Les nouveaux types de données record et table sont déclarés comme des éléments du package. Puis une procédure est déclarée avec comme paramètre le pointeur de type « table » qui pointe le tableau de données.

Exemple-1 : écrire une procédure stockée qui permet d'afficher les valeurs d'une table de données des clients. Cette dernière sera transmise en paramètre ainsi que sa taille**.**

Procedure LesClients (T in out Tab, n integer);

```
create or replace Package test_Pack as
type Str is record (Code clients.idclient%type, Nom
    clients.societe%type, VL clients.Ville%type);
type Tab is table of Str index by Binary_Integer;
```

```
Procedure LesClients (T in out Tab, n integer);
end;
create or replace Package Body test_Pack as
Procedure LesClients (T in out Tab, n integer) as
begin
for i in 1..n loop
    Dbms_Output.Put_Line(T(i).Code||'  '||T(i).Nom||'
    '||T(i).VL);
end loop;
end test;
end;
set Serveroutput on;
declare
i integer:=1;
V test_Pack.Tab;
cursor cur is
select idclient, societe, Ville from clients;
begin
for n in cur loop
V(i).Code:=n.idclient;
V(i).Nom:=n.societe;
V(i).VL:=n.Ville;
i:=i+1;
end loop;
test_Pack. LesClients (V, i-1);
end;
```


Exemple-2 : écrire une fonction stockée qui permet de charger les clients dans une collection (table mémoire) indicée par le code du client (index by varchar2()) puis renvoie un élément de cette collection qui correspond à l'indice du 2ème paramètre de la fonction**.**

function fclients(T IN OUT tab, code clients.codeclient%type) RETURN clients%rowtype

Correction:

```
create or replace PACKAGE pack1 is
type tab is table of clients%rowtype
index by clients.codeclient%type;
function fclients(T IN OUT tab, code clients.codeclient%type)
RETURN clients%rowtype;
end;

create or replace PACKAGE BODY pack1 is
function fclients(T IN OUT tab, code clients.codeclient%type)
RETURN clients%rowtype is
cursor cur is
```

```
select * from clients;
begin
for n in cur loop
T(n.codeclient):=n;
end loop;
RETURN T(code);
end;
end;

set serveroutput on;
declare
S clients%rowtype;
L pack1.tab;
begin
S:=pack1.fclients(L, 'eget');
dbms_output.put_line(S.idclient||'  '||S.codeclient||'
'||S.ville);
end;
```

VIII- Les exceptions (la gestion des erreurs)

La section EXCEPTION permet de gérer les erreurs survenues lors de l'exécution d'un bloc PL/SQL. En utilisant la gestion des exceptions, nous pouvons tester le code et éviter de le quitter brusquement. Lorsqu'une exception survient, un message expliquant sa cause est reçu. Le message PL / SQL Exception se compose de trois parties.

1. Type d'exception
2. Un code d'erreur
3. Un message

Il existe 2 types d'exceptions.

1. Exceptions du système nommé
2. Exceptions définies par l'utilisateur

VIII-1 Exceptions du système nommé

Elles sont automatiquement déclenchées par Oracle lorsqu'un programme enfreint une règle de SGBDR. Il existe certaines exceptions système qui sont fréquemment soulevées, elles sont donc prédéfinies et portent un nom dans Oracle, connu sous le nom d'exceptions système nommées (Named System Exceptions). Ci-dessous un tableau résumant un ensemble d'exception system nommées :

Nom d'Exception	Description	Numéro d'erreur
CURSOR_ALREADY_OPEN	Lorsque vous ouvrez un curseur déjà ouvert.	ORA-06511
INVALID_CURSOR	Lorsque vous effectuez une opération non valide sur un curseur tel que la fermeture d'un curseur, récupérer les données d'un curseur qui n'est pas ouvert.	ORA-01001
NO_DATA_FOUND	Lorsqu'une clause SELECTE... INTO ne renvoie aucune ligne d'une table.	ORA-01403
TOO_MANY_ROWS	Lorsque vous sélectionnez ou récupérez plusieurs lignes dans un enregistrement ou une variable.	ORA-01422
ZERO_DIVIDE	Lorsque vous essayez de diviser un nombre par zéro.	ORA-01476

Exemple :
Supposons qu'une exception NO_DATA_FOUND soit déclenchée dans un processus, nous pouvons écrire un code pour gérer l'exception, comme indiqué ci-dessous.

Syntaxe :

```
BEGIN
  Execution section
EXCEPTION
WHEN NO_DATA_FOUND THEN
 dbms_output.put_line ('SELECT..INTO did not return any row');
 END;
```

VIII-2 Exceptions définies par l'utilisateur

Outre les exceptions système, nous pouvons explicitement définir des exceptions en fonction d'un ensemble de règles définies par l'utilisateur.
Étapes à suivre pour utiliser les exceptions définies par l'utilisateur doivent être :
• explicitement déclarés dans la section de déclaration.

• explicitement mentionnés dans la section de l'exécution.
• gérés en référençant le nom d'exception défini par l'utilisateur dans la section des exceptions.

La syntaxe générale de la partie EXEPTION du bloc PL/SQL est :

```
DECLARE
nom_erreur EXCEPTION;
…
BEGIN
        …
        IF condition THEN
        RAISE nom_erreur; /* On déclenche l'erreur */
        …
        EXCEPTION
        WHEN nom_erreur THEN
        traitement de l'erreur;
        [WHEN OTHERS THEN instr1;
        …..]
END;
```

Exemple : créer un programme qui calcule la quantité totale vendue (QT) de chaque produit. Une exception doit être gérée dans le cas où on trouve un produit dont la QT est supérieure à une quantité limite. Si c'est le cas un message doit d'afficher « Produit à quantité énorme » suivi d'un message d'erreur. Si non on affiche tous les produits avec leur QT.

Réponse :

```
declare
type str is record (code integer, Nom
    Produits.Designation%type, QT integer);
P str;
cursor cur is
select produits.idproduit, designation, sum(quantite) as QT
from produits, lignecommandes
where produits.idproduit=Lignecommandes.Idproduit
group by produits.idproduit, designation;
Ex Exception;
QL integer :=17000;
begin
for n in cur loop
P:=n;
Dbms_Output.Put_Line(P.code||'  '||P.Nom||'    '||P.QT);
if P.QT>QL then
raise Ex;
end if;
```

```
end loop;
Exception
when Ex then
Dbms_Output.Put_Line('Attention Quantite enorme');
end;
```

VIII-3 La Procédure RAISE_APPLICATION_ERROR ( )

C'est une procédure intégrée à Oracle qui permet d'afficher les messages d'erreur définis par l'utilisateur avec le numéro d'erreur négatif dont la plage se situe entre -20000 et -20999.
Cette procédure évite le renvoi des exceptions non traitées, car le numéro d'erreur (inclus dans RAISE_APPLICATION_ERROR) sera communiqué à l'environnement appelant.
Lorsque cette procédure s'exécute, Oracle arrête immédiatement l'exécution du bloc en cours et annule toutes les transactions précédentes qui ne sont pas validées dans le bloc PL/SQL (c'est-à-dire que les instructions INSERT, UPDATE ou DELETE sont annulées).
RAISE_APPLICATION_ERROR est utilisé pour les raisons suivantes,

a) Créer un identifiant unique pour une exception définie par l'utilisateur.
b) Pour que l'exception définie par l'utilisateur ressemble à une erreur Oracle.

La syntaxe générale est :

RAISE_APPLICATION_ERROR (error_number, error_message);

Exemple :
Créer une procédure stockée qui permet d'insérer une nouvelle ligne de commande dans la table « détailsCommandes ». Une exception doit annuler l'insertion et affiche un message d'erreur « Quantité insuffisante » si la Quantité saisie est < 20.

```
create or replace PROCEDURE insertligne(
idligne lignecommandes.idcommande%type,
idcom lignecommandes.idcommande%type,
idprod lignecommandes.idproduit%type,
Q lignecommandes.Quantite%type,
R lignecommandes.remise%type) as
begin
insert into lignecommandes values (idligne,idcom, idprod, Q,
R);
if (Q<20) then
raise_application_error(-20000, 'quantité faible');
end if;
end;

begin
insertligne(30001, 1, 13, 2, 0);
```

```
end;
```

IX- V-5 Les Déclencheurs (TRIGGERs)

Un trigger est une fonction qui doit être exécutée lorsque les opérations INSERT, UPDATE, DELETE sont réalisées. Ces déclenchements peuvent avoir lieu avant ou après ces opérations (BEFORE AFTER).
Pour une requête SQL agissant sur plusieurs lignes simultanément, la clause FOR EACH ROW applique le déclenchement du trigger pour chaque enregistrement ou pour l'ensemble des lignes modifiées.
« :OLD » sont des variables (des enregistrements) qui donnent accès à la valeur avant la mise à jour (UPDATE). Dans le cas de l'insertion cette valeur est NULL.
« :NEW » sont des variables (des enregistrements) qui donnent accès à la valeur après la mise à jour (UPDATE). Dans le cas de la suppression cette valeur est NULL.
Une fois codé puis compilé, le déclencheur est stocké dans la base et s'excite chaque fois que l'évènement déclenchant s'exécute. Il est possible de contrôler les conditions d'exécution avec des clauses telles que
IF INSERTING THEN
...
ELSIF UPDATING THEN
...
ELSIF DELETING THEN
...

syntaxe :
CREATE OR REPLACE TRIGGER nom_TRigger
BEFORE INSERT (Update) [DELETE] ON Table
for EACH ROW
BEGIN (debut de trigger)
declare
....
begin
……
end;
END (Fin Trigger);

Exemple : créer un trigger qui permet de vérifier la disponibilité d'un produit avant l'insertion d'une nouvelle ligne de commande.

CREATE OR REPLACE TRIGGER TRIGGER1
BEFORE INSERT ON détailsCommandes

```
for EACH ROW
BEGIN
declare
  Qt number;
  begin
  select PRODUITS."UNITÉSENSTOCK"+"UNITÉSCOMMANDÉES" into Qt
  From produits
  where RéfProduit=:New.RéfProduit;
  if :New.Quantité>Qt then
  RAISE_APPLICATION_ERROR(-20000,'Quantité en stock insuffisante');
  end if;
  end;
END;
```